Recitation 1 — Parenthesis Matching and SML Review (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

August 29, 2012

Welcome to 210! This recitation is aimed at helping you shake off some sand from the beach and getting you started on Homework 1, which will be released later today. We will be using SML/NJ as our default programming language, which you should be familiar with if you have taken 15-150 previously. For those who haven't (graduate students, for example), we will be holding crash courses today and tomorrow night in GHC 5201 (see Piazza for updates). We will expect you to write clean and readable (self-documenting!) code as well as mathematical proofs.

1 Administrivia

Where is my assignment? We will be distributing the assignments for this course through a read-only `git`[1] repository. To start you off, we've put together a handout on `git` commands, with pointers to more advanced features:

`http://www.cs.cmu.edu/~15210/resources/git.pdf`

which will also be linked from the Resources page.

When are Office Hours? We want to accomodate your schedules, so please fill out the poll on Piazza. When they have been finalized, Office Hours will be posted on the course webpage at

`http://www.cs.cmu.edu/~15210/staff.html`

These times are subject to change. If you have time conflicts and cannot attend any of the listed office hourse, please contact one of the course staff.

What is my grade? If you want to know your grades, visit the Gradebook page on the course website and follow the instructions there. You will need to log in with your WebISO credentials.

When are the ML tutorials? For those students with no prior experience with programming in ML, there will be tutorials tonight and tomorrow at 8pm in the GHC 5201 computer clusters. For a preview, see the brief tutorial and book referenced under **SML Style Guide** on the Resources page of the course website.

[1] `git` is a fully distributed version control system, initially developed for Linux kernel development. Since nobody reads footnotes, we won't go into any more detail here.

2 Let's Begin!

We'll begin with a running example: the parenthesis matching problem. We define it as follows:

- **Input:** a `char` sequence `s : char Sequence.seq`, where each s_i is either an "(" or ")". For instance, we could get a parenthesis-matched sequence

  $$s = \langle (, (,), (,),) \rangle$$

  or a non-matching one

  $$t = \langle), (,), (,),) \rangle$$

- **Output:** `true` if `s` represents a parenthesis-matched string and `false` otherwise. In the above examples, the algorithm should output `true` on input s and `false` on input t.

To simplify the presentation, we will be working with a `paren` data type instead of characters. Specifically, we will write a function `match` of type `paren Sequence.seq -> bool` that determines whether the input is a *well-formed parenthesis expression* (i.e., it is a parenthesis-matched sequence). The type `paren` is given by

```
datatype paren =
     OPAREN
   | CPAREN
```

where `OPAREN` represents an open parenthesis and `CPAREN` represents a close parenthesis.

So, how would we go about solving this problem? Lets begin with a simplest sequential solution and work our way to a work-optimal parallel solution.

2.1 Sequence `iter`

As in 15-150, you will be making extensive use of a `SEQUENCE` library throughout this course. For the current problem, we'll use the function `iter` (for iterate) from the sequence library. It has the following type:

```
val iter :  ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b
```

If `f` is a function, `b` is a value, and `s` is a sequence, then `iter f b s` iterates f with left association on s using b as the base case. You may think of f as a state transition function and b as a base state.

2.2 Back to Parentheses

How can we use this to solve the parenthesis matching problem? As we iterate across the sequence, we can keep track of the number of open parentheses that we have seen so far and subtract from this number when we find a closing one. When we reach the end of the sequence we should have 0, given that we started at 0.

Is this sufficient?

No, if the number goes below zero at any point, then when we know we can't possibly have a well-formed parenthesis expression—we'll designate a special state to represent this outcome.

What type should we use to represent the state?

You should be able to figure it out from the following code:

```
fun match s =
    let
      fun check (NONE, _) = NONE
        | check (SOME c, OPAREN) = SOME (c+1)
        | check (SOME 0, CPAREN) = NONE
        | check (SOME c, CPAREN) = SOME (c-1)
    in
      (iter check (SOME 0) s) = (SOME 0)
    end
```

You can show that this solution has $O(n)$ work and span, where n is the length of the input sequence. *How can we make it more parallel?*

3 Divide and Conquer

As you have already seen in previous classes, divide and conquer is a powerful technique in algorithms design that often leads to efficient parallel algorithms. A typical divide and conquer algorithm consists of 3 main steps (1) divide, (2) recurse, and (3) combine.

To follow this recipe, we first need to answer the question: how should we divide up the sequence? We'll first try the simplest choice, which is to split it in half—and attempt the merge their results somehow. This leads to the next question: what would the recursive calls return?

Let's try returning whether the given sequence is well-formed. Clearly, if both s_1 and s_2 are well-formed expressions, s_1 concatenated with s_2 must be a well-formed expression. However, we could have s_1 and s_2 such that neither of which is well-formed but $s_1 s_2$ is well-formed (e.g., "(((" and ")))"). This is not enough information to conclude whether $s_1 s_2$ is well-formed.

We need more information from the recursive calls. You are probably already familiar with a similar situation from mathematical induction—you often need to strengthen the inductive hypothesis. We'll crucially rely on the following observations (which can be formally shown by induction):

Observation 3.1. *If s contains "()" as a substring, then s is a well-formed parenthesis expression* ***if and only if*** *s' derived by removing this pair of parenthesis "()" from s is a well-formed expression.*

Observation 3.2. *If s does* ***not*** *contain "()" as a substring, then s has the form "$)^i($". That is, it is a sequence of close parens followed by a sequence of open parens.*

That is to say, on a given sequence s, we'll keep simplifying s *conceptually* until it contains no substring "()" and return the pair (i, j) as our result. This is relatively easy to do recursively. Consider that if

$s = s_1 s_2$, after repeatedly getting rid of "()" in s_1 and separately in s_2, we'll have that s_1 reduces to "$)^i($" and s_2 reduces to "$)^k($" for some $i, j, k, \ell \in \mathbb{Z}_+ \cup \{0\}$. To completely simplify s, we merge the results. That is, we merge "$)^i($" with "$)^k($". The rules are simple:

- If $j \leq k$ (i.e., more close parens than open parens), we'll get "$)^{i+k-j}($".
- Otherwise $j > k$ (i.e., more open parens than close parens), we'll get "$)^i($".

This directly leads to a divide and conquer algorithm.

3.1 How to split a sequence in half?

The sequence library we give you provides a conceptual view of sequences called `treeview` that lends itself particularly well to divide-and-conquer algorithms. For those of you who have used `listview` in 15-150, this concept will be very familiar. To review, we have a data type `'a treeview` defined as follows:

```
datatype 'a treeview =
     EMPTY
   | ELT of 'a
   | NODE of ('a seq * 'a seq)
```

The function `showt` provides a means to examine the sequence in the `treeview`:

```
val showt :  'a seq -> 'a treeview
```

Essentially, `showt s` splits the sequence in approximately half and returns both halves as sequences, provided that the input sequence has length at least 2. The two base cases are for empty and singleton sequences.

3.2 Implementing the algorithm in `treeview`

To make it more obvious which calls are being made in parallel, we will also introduce a function

```
par :  (unit -> 'a) * (unit -> 'b) -> 'a * 'b
```

If you run `par (f, g)`, this construct allows you to execute the two functions `f` and `g` in parallel.

```
fun match s =
    let
      fun match' s =
          case showt s
           of EMPTY => (0, 0)
            | ELT OPAREN => (0, 1)
```

```
          | ELT CPAREN => (1, 0)
          | NODE (L, R) =>
            let
              val ((i, j), (k, l)) =
                par (fn () => match' L, fn () => match' R)
            in
              if j > k then (i, l + j - k)
              else (i + k - j, l)
            end
in
  match' s = (0, 0)
end
```

Running Time Analysis: Let's assume that `showt s NONE` takes $O(\log n)$ work and span on any sequence of length n. We can formulate the work and span recurrences as follows:

$$\begin{aligned} W(n) &= 2 \cdot W(n/2) + W_{showt}(n) = 2 \cdot W(n/2) + O(\log n) \\ S(n) &= S(n/2) + S_{showt}(n) = S(n/2) + O(\log n). \end{aligned}$$

It is not too hard to see that $S(n)$ is $O(\log^2 n)$

$$\begin{aligned} S(n) &= \log(n) + \log(n/2) + \ldots + \log(1) \\ &\leq \underbrace{\log(n) + \log(n) + \ldots + \log(n)}_{\log n \text{ times}} \\ &\in O(\log^2 n) \end{aligned}$$

It is a little more work to see $W(n) = O(n)$.

Lemma 1

$$\sum_{i=0}^{\lg n} 2^i = 2n - 1$$

Proof. Take $S = \sum_{i=0}^{\lg n} 2^i = 2^0 + 2^1 + \cdots + 2^{\lg n}$. We can get the closed form directly using a formula for geometric sums, but it's easy to derive:

$$\begin{aligned} 2S = &\quad 2^1 + 2^2 + \cdots + 2^{\lg n} + 2^{\lg(n+1)} \\ S = \; 2^0 + &\quad 2^1 + 2^2 + \cdots + 2^{\lg n} \end{aligned}$$

Subtracting S from $2S$, the terms in the middle cancel out and we're left with

$$S = 2^{\lg(n+1)} - 2^0 = 2n - 1$$

□

Lemma 2

$$\sum_{i=0}^{\lg n} i2^i = 2n\lg n - (2n-2)$$

Proof. Take $S = \sum_{i=0}^{\lg n} i2^i = 0\cdot 2^0 + 1\cdot 2^1 + 2\cdot 2^2 + \cdots + \lg n 2^{\lg n}$:

$$\begin{aligned} S = \quad & 1\cdot 2^1 + \quad 2\cdot 2^2 + 3\cdot 2^3 + \cdots + \lg n 2^{\lg n} \\ 2S = \quad & \qquad\quad 1\cdot 2^2 + 2\cdot 2^3 + \cdots + (\lg n - 1)2^{\lg n} + \lg n 2^{\lg n+1} \end{aligned}$$

Subtracting $2S$ from S, we can apply Lemma 1 from above

$$\begin{aligned} -S &= 2^1 + 2^2 + \cdots + 2^{\lg n} - \lg n 2^{\lg n+1} \\ -S &= (2n-2) - \lg n 2^{\lg n+1} = (2n-2) - 2n\lg n \\ S &= 2n\lg n - (2n-2) \end{aligned}$$

□

Theorem

$$W(n) = 2W(n/2) + O(\lg n) \in O(n)$$

$$\begin{aligned} W(n) &= \sum_{i=0}^{\lg n} 2^i \left(a\lg(n/2^i) + b\right) \\ &= a\sum_{i=0}^{\lg n} 2^i \left(\lg n - i\right) + b\sum_{i=0}^{\lg n} 2^i \\ &= a\lg n \sum_{i=0}^{\lg n} 2^i - a\sum_{i=0}^{\lg n} i2^i + b\sum_{i=0}^{\lg n} 2^i \\ &= a\lg n(2n-1) - a(2n\lg n - (2n-2)) + b(2n-1) \qquad \textit{(Lemma 1\&2)} \\ &= 2an\lg n - a\lg n - 2an\lg n + 2an - 2a + 2bn - b \\ &= (a+b)(2n-1) - a\lg n - a \in O(n) \end{aligned}$$

■

4 To Be Continued...

Homework 1 will be released later today, and due next Wednesday. There will be a proof of correctness, but note that we will not be looking for a step-by-step code evaluation trace. You should be familiar with SML evaluation by now, so we'll be more interested in a higher-level discussion of the algorithm itself. Please check the course website for updates on office hours if you have trouble.